Projet Logiciel Transversal

Battle city

LI Yijie

Talbe des matières

1 Objectif

1.1 *Présentation générale*

Le jeu Battle City consiste en une bataille de tanks. Le terrain est composé d'obstacles(comme un labyrinthe). Chaque tank est muni d'un canon et doit tirer et toucher les tank adverses.

1.2 *Règles du jeu*

Le jeur se joue par un seul joueur.

Pour diriger le tank, il faut utiliser les flèches directionnelles Haut Bas Gauche Droite pour se diriger le tank de joueur, et le boutton F pour tirer. Le joueur doit tuer toutes les tanks des ennemis. Les tanks adverses jouent automatiquement(IA) et ils essaient de détruire la base de joueur(un oiseau).
Le jeu finit si la base est détruit ou le joueur est touée.

2 Description et conception des états

2.1 *Description des états*

Le jeu est formé par un ensemble d'éléments fixes (mur en bois et mur en fer) et un ensemble d'éléments mobiles (tanks). Tous les éléments les coordonnées et identifiant de type d'élément.

Le class **StaticElement**: il comporte des murs en différentes types (wood,iron,sea)

Le class **MobileElement**: il représente les unités de tank. Ces unités partagent beaucoup d'attributs et de fonctions. Elles ont toutes de la vie, une vitesse de déplacement.

A l'ensemble des éléments statiques et mobile, on rajoute quelque propriétés:

Epoque: il représente temps correspondant à l'état du jeu.

Vitesse: le nombre d'époque par seconde.

Computer d'unités: le nombre d'unités d'une même équipe

2.2 *Conception logiciel*

Dans le diagramme des classes pour les états (Figure 1), on a quelque classes:

Class Element: Toute les classes filles d'Element représenter les catégories d'élément.

La fabrication d'éléments: On utilise la classe ElementFactory pour fabriquer d'éléments.

Conteneurs d'éléments: On met toutes les éléments dans ElementList et ElementGrid. ElementList a une list des toutes les éléments et ElementGrid peut gérer une grille.

2.3 *Conception logiciel: extension pour le rendu*

Observateurs de changements. Dans le diagramme de classes, on présente les classes permettant a des tiers de réagir lorsqu'un événement se produit dans l'un des éléments d'état.
Les observateurs implantent l'interface StateObserver pour être avertis des changements de propriétés d'état. Pour connaître la nature du changement, ils analysent l'instance de StateEvent. La conception de ces outils suit le patron Observer.

2.4 *Conception logiciel: extension pour le moteur de jeu*

Dans quelque classes par rapport à Element, on ajoute des méthodes de clone() et equals().

Méthode clone(): elle peut faire la copie de l'instance
Méthode euqals(): elle peut faire la comparaison entre deux instances.

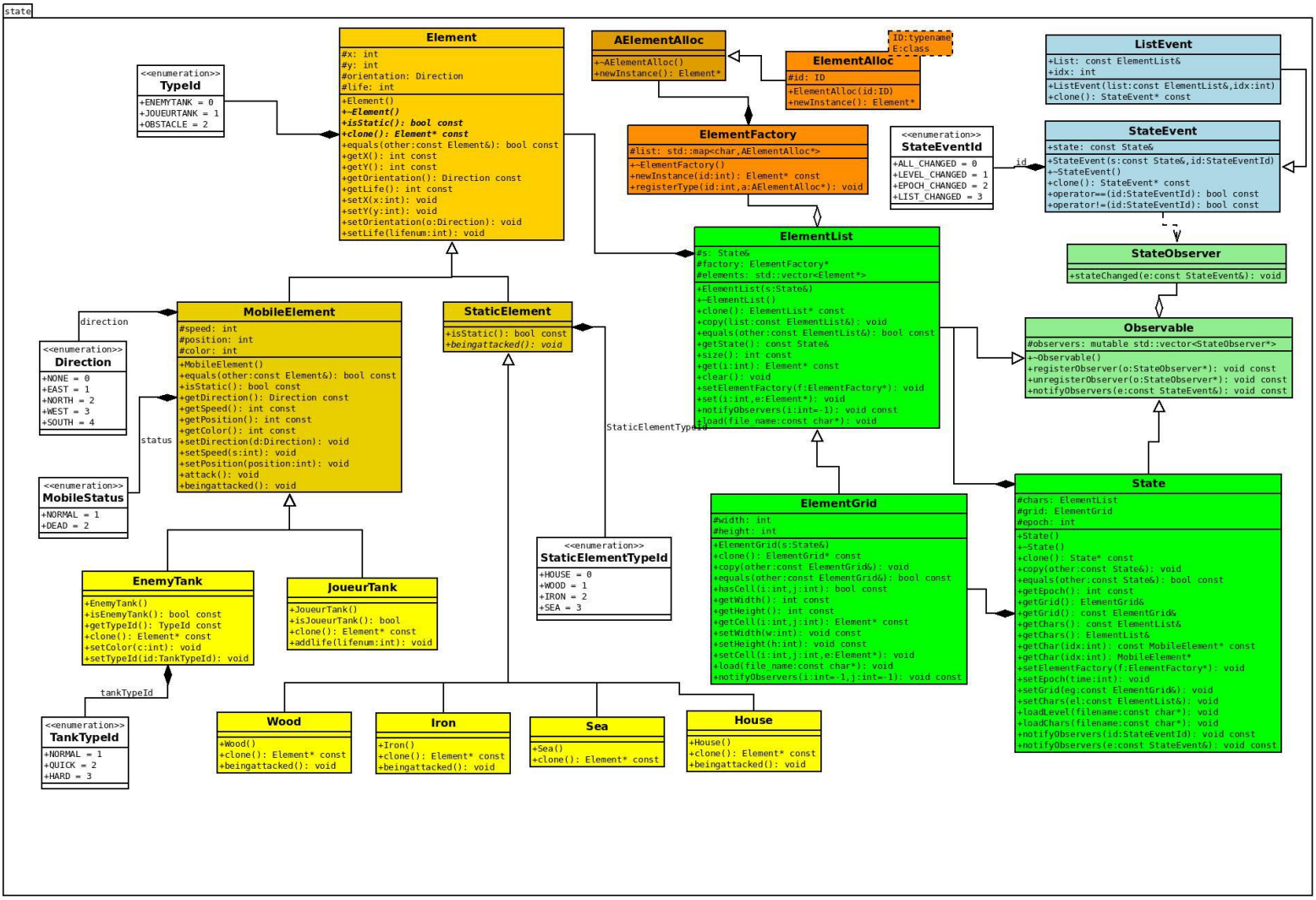

Figure1: Diagramme de classes de l'état

3. Rendu : Stratégie et Conception

3.1 *Stratégie de rendu d'un état*

Pour le rendu d'un état, nous avons opté pour une stratégie assez bas niveau, et relativement proche du fonctionnement des unités graphiques. En effet, les cartes graphiques actuelles, tout comme celles des années 80, sont plus efficace si le CPU prépare les éléments à rendre au sein de structures élémentaires, avant de tout envoyer au GPU.

Plus précisément, nous découpons la scène à rendre en plans (ou « layers ») : un plan pour le niveau (mur, pastilles, etc.), un plan pour les éléments mobiles (pacman, fantômes) et un plan pour les informations (vies, scores, etc.). Chaque plan contiendra deux informations bas-niveau qui seront transmises à la carte graphique : une unique texture contenant les tuiles (ou « tiles »), et une unique matrice avec la position des éléments et les coordonnées dans la texture. En conséquence, chaque plan ne pourra rendre que les éléments dont les tuiles sont présentes dans la texture associée.

3.2 *Conception logiciel*

Le diagramme des classes pour le rendu est en Figure 4.

On utilise la bibliothèque SFML, on a choisi d'implémenter l'interface sf::drawable à render::scene et ses layers. Cette implémentation permet de simplifier les appels graphiques.

Layer

Un layer(plan) est un objet réalisant le lien entre une liste d'éléments et leur position spatiale dans la fenêtre ainsi que leur représentation(texture). Layer hérite de sf::drawable permet de dessiner out le layer.

Scene

Elle contient tous les plants, et son implémentation de draw() appelle les draw() de ses layers.

Elle possède une fonction update(ElementList), qui est appelée lors d'une modification de l'état(pattern Observer)

Tile

Cet objet est simplement une représentation de la clef et les informations de texture associée.

3.3 *Conception logiciel: extension pour les animations*

Animations. Les animations sont gérées par les instances de la classe Animation. Chaque plan tient une liste de ces animations, et le client graphique fait appel aux méthodes de mise à jour pour faire évoluer ses surfaces.
Pour le cas des animations de mouvement, par exemple lorsqu'un tank se déplace, on a ajouté toutes les informaitons permettant d'afficher le déplacement sans dépendre de l'état.

3.4 *Ressources*

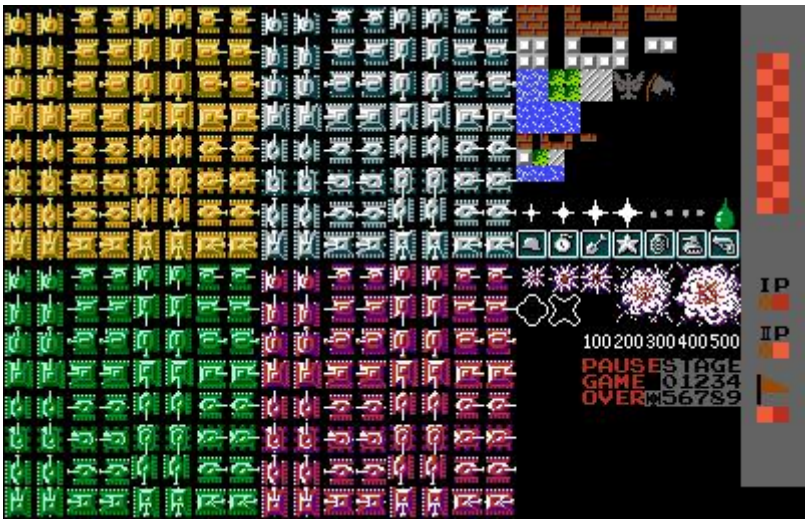

Figure 2: Exemple de texture

3.5 Exemple de rendu

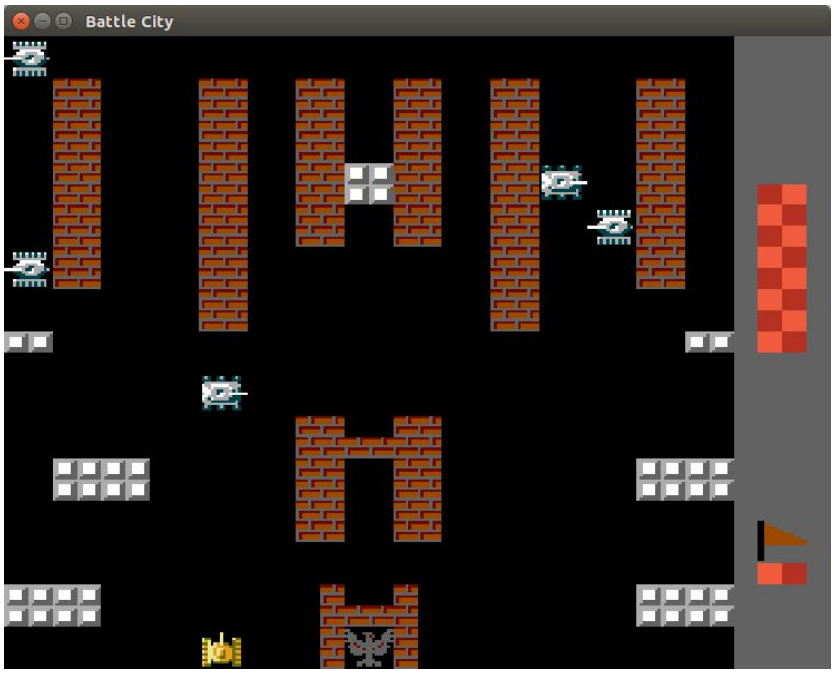

Figure 3 : Exemple de rendu

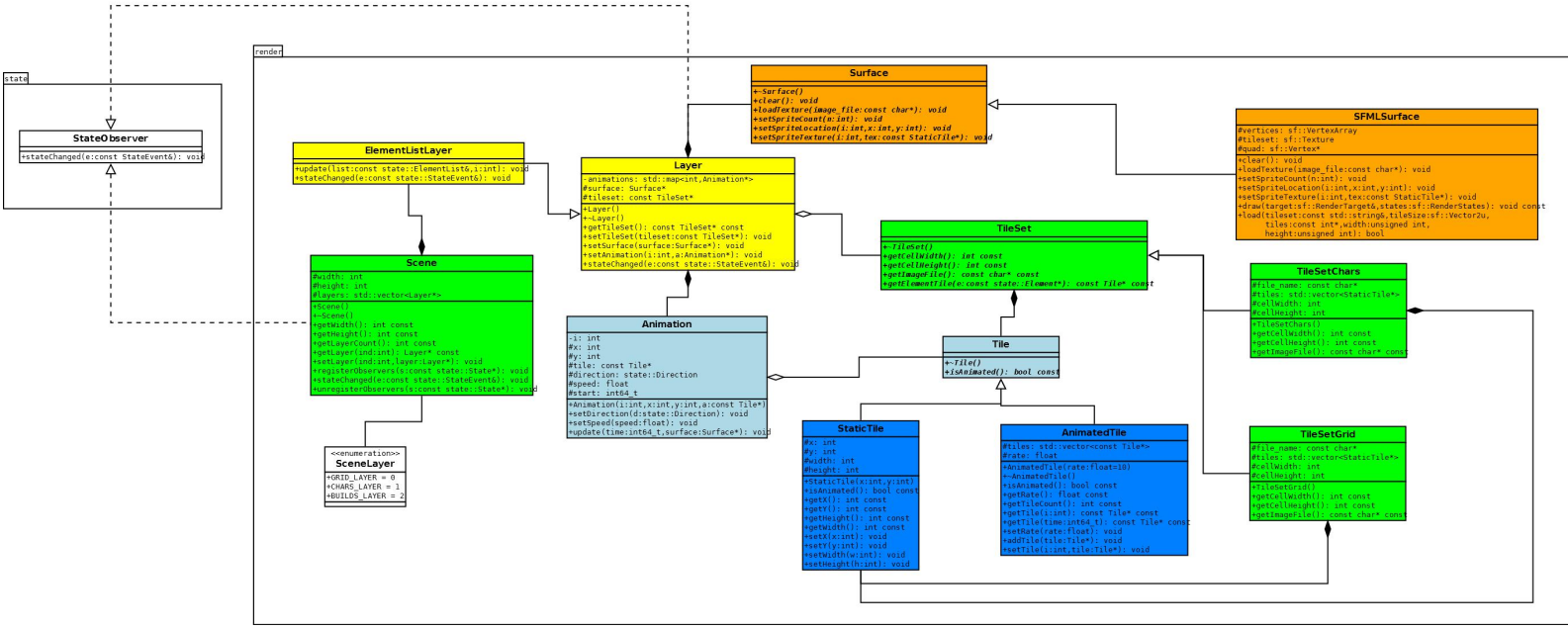

Figure 4 : Diagramme de classes du moteur de rendu

4. Règle de changement d'état et moteur de jeu

4.1 *Epoques*

Il y a un changement d'époque à chaque action construite par le moteur de jeu. Une liste d'action est crée.

4.2 *Changements extérieurs*

Les changements extérieurs sont provoqués par des commandes extérieurs, comme la pression sur une touche ou le clic sur un bouton de la souris.
Ces commandes peuvent être:

- le déplacement d'une unité
- le chargement d'un mode de jeu.
- le chargement d'une partie

4.3 *Changements autonomes*

A terme, plusieurs changements s'effectueront automatiquement à chaque tour.

- le changement d'état d'une unité.
- lorsqu'une unité se déplace ou attaque, on décremente ses attributs, soit respectivement ses points de déplacement ou ses munitions, ses points de vie.
- appliquer les règles de mouvement pour toutes les unités

4.4 *Conception logiciel*

Le diagramme des classes pour le moteur du jeu est présenté en Figure 5. L'ensemble du moteur de jeu repose sur un patron de conception de type Command, et a pour but la mise en oeuvre différée de commandes extérieures sur l'état du jeu.

Classes Command : Cette classe représente les différentes types de commande qui peuvent être effectués par le joueur ou l'IA. L'ensemble des commandes est stocké à l'aide de la classe CommandSet.

Engine : Cette classe est la plus importante concernant le moteur de jeu. Elle se charge transformer les commandes stockées en actions à effectuer. Ces actions sont stockées avec la classe ActionList, actions représentées par des classes fille de la classe mère abstraite Action

Action : Cette classe mère et ses classes dérivées se contentent d'effectuer les modifications demandées, comme les classes fille MoveCharacter, qui permet de déplacer une unité tout en lui affectant une orientation, ou la classe IncEpoch qui incrémente la variable epoch correspondant à l'époque pour un état défini.

4.5 *Conception logiciel : extensin pour l'IA*

Record : Ces mécanismes nous permettent d'enregistrer toutes les actions, et par conséquence de rejouer, à l'endroit ou à l'envers, tout ce qui a été enregistré. Pour cela, la classe possède une liste d'ActionList, ce qui permet, à partir d'un état initial, d'arriver à l'état final en ré-effectuant toutes les actions faites par les joueurs (IA ou non).

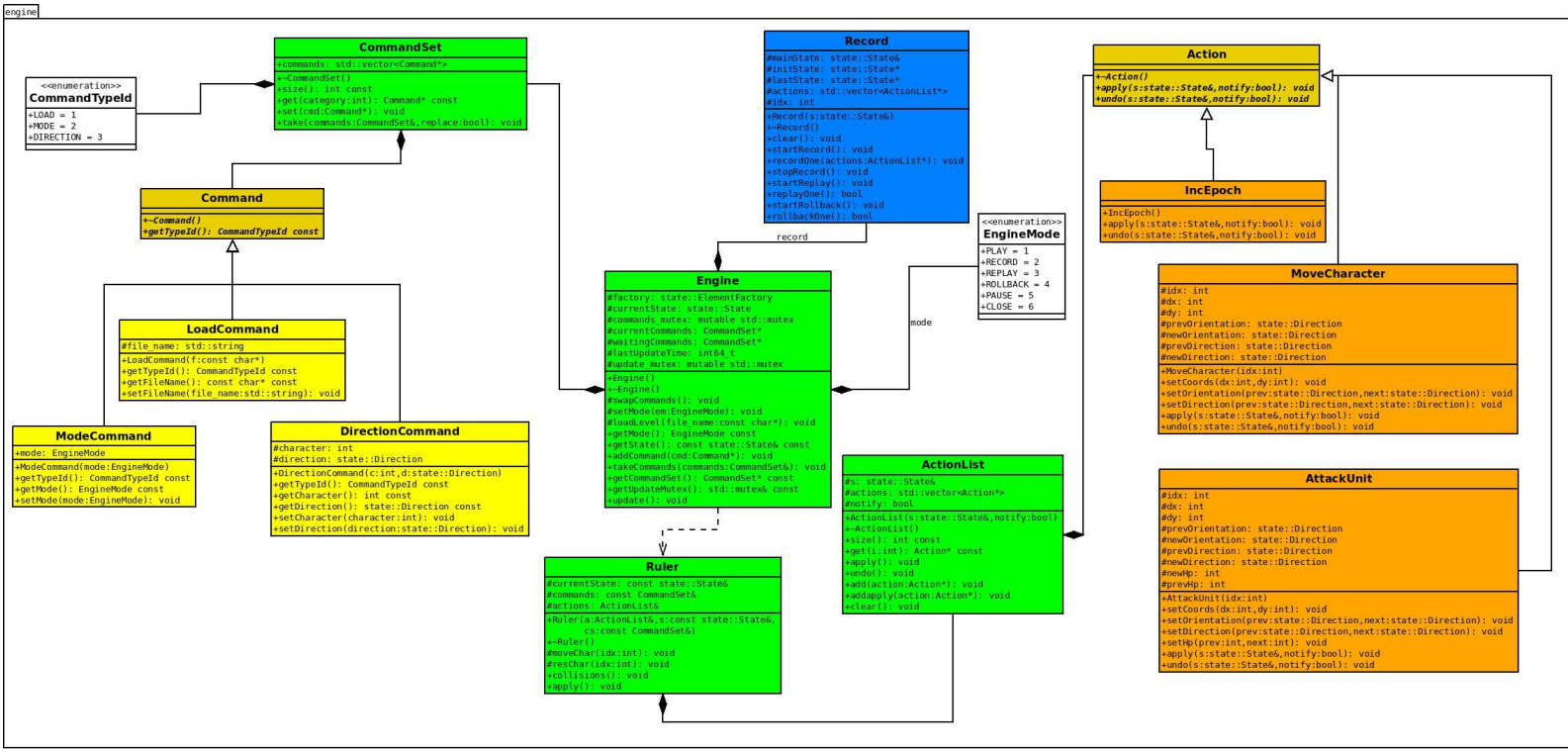

Figure 5 : Diagramme de classes du moteur de jeu

5 Intelligence Artificielle

Il y a plusieurs IA correspondant à la difficulté que le joueur choisit. Il y a l'IA simple qui agit aléatoirement, puis l'IA intermédiaire qui prend des décisions logiques et enfin l'IA avancée, qui analyse la situation.

5.1 *Stratégie*

5.1.1 Intelligence minimale

L'IA simple se contente d'avancer ses unités à des positions aléatoires.

5.1.2 Intelligence basée sur des heuristiques

Dans le but d'offrir un comportement plus efficace, un ensemble d'heuristiques sont proposées.

La principale amélioration concerne le choix de déplacement d'IA. Jusqu'alors, des mouvements comportaient une partie aléatoire lors de ses déplacements et tours les tanks.

Ainsi dans un souci d'authenticité, la version heuristique s'appuie sur des cartes de distance permettant à l'IA de converger efficacement vers la cible(la maison de joueur)

Dés que l'IA apparaîtra, il sera à portée sur le fond de map pour détruire la maison de joueur. Pour réaliser cette fonction, parmi de trois directions génères aléatoires, l'IA va exécuter une fonction de la direction au sud.

5.2 *Conception logiciel*

Classes AI : toutes les formes d'intelligence artificielle implémentent la classe abstraite AI. Le rôle de ces classes est de fournir un ensemble de commandes à transmettre au moteur de jeu. Notons qu'il n'y a pas une instatnce par joueur, mais qu'une instance doit fournir les commandes pour tous les tanks.
Cette class implante l'intelligence minimale.

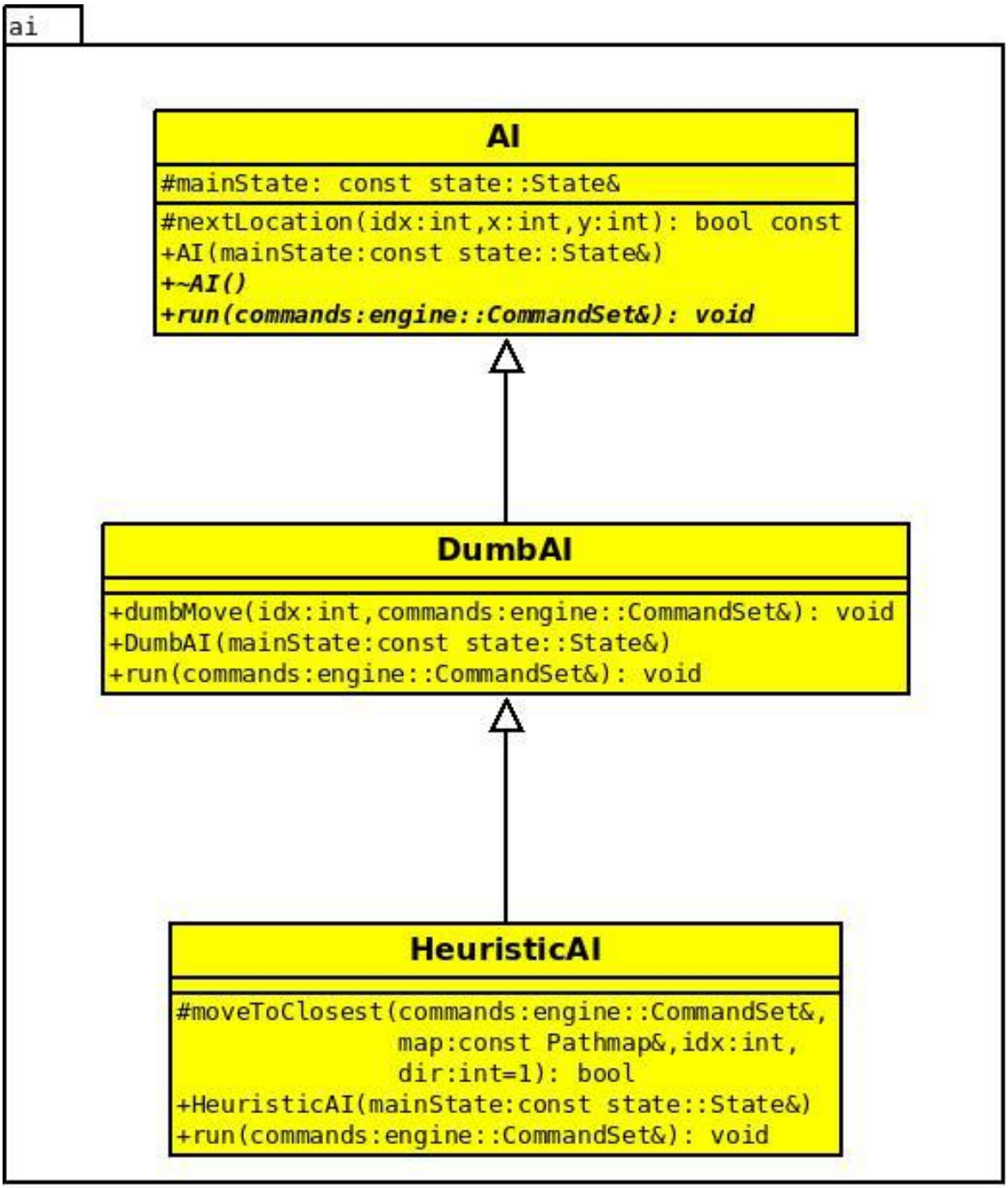

Figure 6: Diagramme de classe de l'IA

6. Modularisation

6.1 Organisation des modules

6.1.1 Répartition sur différent threads

Le but est de créer trois threads sur le mouvement des trois tanks ennemis.

Moteur de jeu On ulitise trois listes de commands(classe Surface): chacun pour la commande de chque tank et il sert de buffer si le moteur reçoit des commandes alors que l'état est en train d'être mis à jour.

6.2 API Web

6.2.1 Services des connexion

PUT

Requête: méthode HTTP et URI: PUT/connexion

```
type:"object",
properties:{
    "joueur": {type:int},
},
required:["joueur"]
```

Réponse: Cas où la connextion est validée:
(a) Status : 200(=OK)
(b) Données:

```
type:"object",
properties:{
    "id": {type:int},
},
required:["id"]
```

Cas où la connextion n'est pas validée: Status 403(=FORBIDDEN), Pas de données.

DELETE

Requête: méthode HTTP et URL: DELETE/connexion/id. Pas de données.

Réponse:

si elle est validée: Status :200(=OK). Pas de données

si elle n'est pas validée : Status :403(=FORBIDDEN). Pas de données

6.2.2 Services des commandes

PUT

Requête: méthode HTTP et URI: PUT/commande/

Données:

```
type:"object",
properties:{
    "commandtype":{
    "type":"string",
    "pattern":"(load|mode|direction)""
    },
}
required:["load","mode","direction"]
```

Réponse
Cas où la commande n'est pas validée: Status: 400(=BAD_REQUEST), Pas de données.

GET

Requête: méthode HTTP et URI: GET/commande/, Pas de données.
Réponse:

```
type:"object",
properties:{
    "commandtype":{
    "type":"string",
    "pattern":"(load|mode|direction)""
    },
}
required:["load","mode","direction"]
```

Cas où la commande n'existe pas : Status: 404(=NOT FOUND), Pas de données.

6.3 Conception logiciel

Classe Server. Cette classe représente le moteur de jeu, donc elle contient un attribut de type Engine, ce qui va permettre la modification de l'état du jeu. La classe mère Server contient les différentes données que les serveurs ont en commun. Si le serveur est sur la machine courante, on utilise la classe LocalServer.

Classe Client. Cette classe repésente le moteur de rendu. La classe mère Client contient toutes les informations que les clients ont en commun. Dans ce cas, puisqu'on utilise la librairie SFML, on utilisons la classe SFMLClient.

6.4 Conception logiciel: extension réseau

Ici, on choisit d'utiliser le service Web REST pour plusieurs raisons. La première est qu'avec les méthodes HTTP, **GET**, **PUT**, **POST** et **DELETE,** on peut récuperer les ressources simplement et "intuitivement". Ensuite le format de répresentation des ressources est très proche de XML, ce qui permet une utilisation optimale.

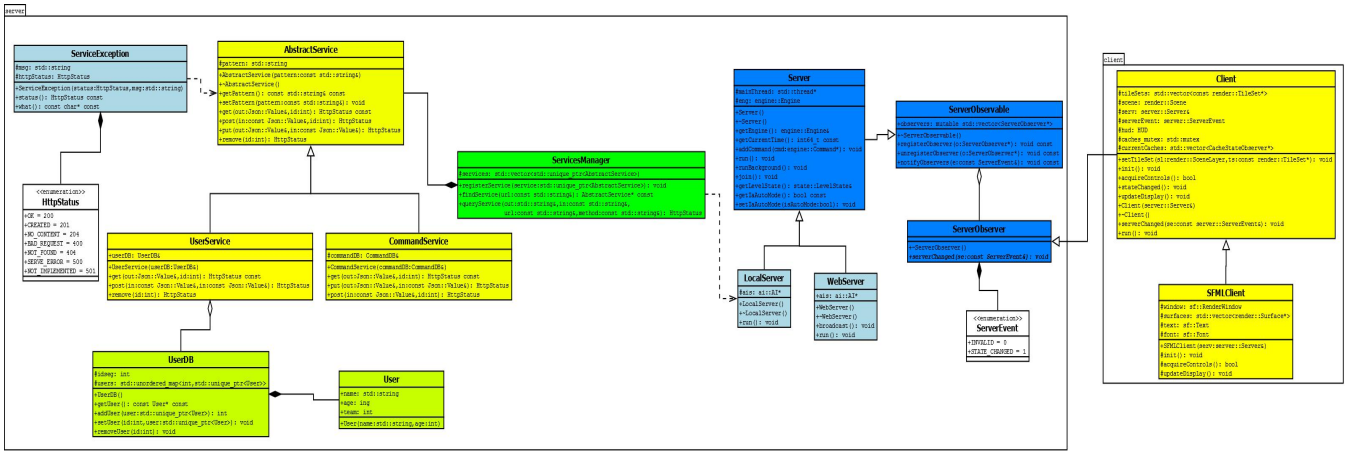

Figure 7: Diagramme de classe pour Client et Serveur